Cross-Language Interoperability of Heterogeneous Code

Athanasios Stratikopoulos
The University of Manchester
Manchester, United Kingdom
{first}.{last}@manchester.ac.uk

Florin Blanaru*
OctoML
Seattle, United States
fblanaru@octoml.ai

Juan Fumero
The University of Manchester
Manchester, United Kingdom
juan.fumero@manchester.ac.uk

Maria Xekalaki
The University of Manchester
Manchester, United Kingdom
maria.xekalaki@manchester.ac.uk

Orion Papadakis
The University of Manchester
Manchester, United Kingdom
orion.papadakis@manchester.ac.uk

Christos Kotselidis
The University of Manchester
Manchester, United Kingdom
christos.kotselidis@manchester.ac.uk

ABSTRACT

In recent years, the Java Virtual Machine has evolved from a cross-ISA virtualization layer to a system that can also offer multilingual support. GraalVM paved the way to enable the interoperability of Java with other programming languages, such as Java, Python, R and even C++, that can run on top of the Truffle framework in a unified manner. Additionally, there have been numerous academic and industrial endeavors to bridge the gap between the JVM and modern heterogeneous hardware resources. All these efforts beacon the opportunity to use the JVM as a unified system that enables interoperability between multiple programming languages and multiple heterogeneous hardware resources.

In this paper, we focus on the interoperability of code that accelerates applications on heterogeneous hardware with multiple programming languages. To realize that concept, we employ TornadoVM, a state-of-the-art software for enabling various JDK distributions to exploit hardware acceleration. Although TornadoVM can transparently generate heterogeneous code at runtime, there are several challenges that hinder the portability of the generated code to other programming languages and systems. Therefore, we analyze all challenges and propose a set of modifications at the compiler and runtime levels to constitute Java as a prototyping language for the generation of heterogeneous code that can be used by other programming languages and systems.

CCS CONCEPTS

• **Software and its engineering → Language features**.

KEYWORDS

code interoperability, programming languages, heterogeneous hardware

*The work was done when employed at the University of Manchester.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX,
,
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Athanasios Stratikopoulos, Florin Blanaru, Juan Fumero, Maria Xekalaki, Orion Papadakis, and Christos Kotselidis. 2023. Cross-Language Interoperability of Heterogeneous Code. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Previous research work has showcased how Java and particularly the Java Virtual Machine (JVM) can be used as a platform that facilitates the implementation of new programming languages [11]. The emergence of GraalVM along with its unified approach to reuse the optimizing JIT compiler for multiple programming languages, have spearheaded interoperability across multiple programming languages, such as Java, Python, R, Ruby, and C++ (via Sulong). The polyglot support can be useful in software domains (e.g., statistical analysis, machine learning, etc.), which employ a wide range of programming languages and libraries [10].

In the mean time, several academic and industry efforts [1, 4–6, 9] have examined ways to bridge the gap between the JVM, which executed exclusively on CPUs, and hardware acceleration. Although hardware acceleration is not natively supported by the JVM, there are technologies, such as TornadoVM [4], that aim to offer a programmer-friendly way for Java programmers to access modern hardware resources, such as GPUs, and FPGAs. TornadoVM [3] exposes a hardware-agnostic API to Java programmers and transparently generates heterogeneous code via a JIT optimizing compiler. TornadoVM includes three compiler backends that can generate either source code (i.e., OpenCL, PTX), or binary code (i.e., SPIR-V). Although Java programmers that use TornadoVM are abstracted from the knowledge of heterogeneous programming models (e.g., OpenCL, CUDA, OneAPI), the heterogeneous code that is generated can be executed only within TornadoVM. This is due to numerous challenges, with the most severe one lying on the TornadoVM memory management system which allocates a memory space in device that enacts as a device memory heap [2]; similar to the JVM heap.

In this paper, we discuss the lessons learned from the emerging challenges and we present our work-in-progress to enable interoperability of the generated heterogeneous code with other programming languages and systems, besides Java. In detail, this paper makes the following contributions:

- It analyzes the challenges that block the portability of TornadoVM generated code with other programming languages and systems.

- It presents our work-in-progress system, which proposes a new mode, called "code-interoperability-mode", as a mean to alleviate the emerging challenges and emit code that has cross-language interoperability.
- It beacons how the JVM and Java can be used as a unified platform and programming language for prototyping heterogeneous code that can run in parallel on various types of hardware and from different programming languages.

2 CHALLENGES FOR CROSS-LANGUAGE INTEROPERABILITY

This section presents an overview of the challenges posed within the JVM and TornadoVM with regard to the portability of heterogeneous code to other programming languages and systems. The former two challenges concern the JVM as a platform, whereas the latter two are specific to TornadoVM and the interoperability with other programming languages.

2.1 Memory Management

JVM is a managed runtime environment that offers automatic memory management. Java programmers are not required to explicitly perform operations such as the *allocation* and *release* of memory space. Instead, the platform can handle these operations in an automatic manner via its Memory Management, and by employing Garbage Collection (GC). Figure 1 illustrates the way that memory is managed between the JVM and TornadoVM (any version prior to v0.14). As shown in the figure, the JVM offers the `heap` memory space, in which all the objects that are created by an application reside. The objects are stored in the heap until they are no longer referenced, and they are garbage collected. The size of the JVM heap along with the growth rate of the utilized memory of an application are factors that can increase the frequency of garbage collection. Additionally, the JVM allows programmers to allocate and use `off-heap` memory. The memory space that operates off heap does not fall into the garbage collection, and thus, it is not managed by the runtime but has to be declared explicitly.

Similarly to JVM, TornadoVM (any version prior to v0.14) allocates a memory space on each targeted hardware device that acts as a device memory heap [2]. That memory space is a replica of the heap memory space that is used by a Java application, and it is transferred on the device memory (i.e., DDR memory) in order to allow any heterogeneous code that executes on the device to access the data in the heap. However, an emerging challenge is that the maximum size of the allocation of a single memory space on the device memory is significantly lower than the physical memory capacity, thereby resulting in memory underutilization. For instance, the maximum size of a permitted allocation in OpenCL is a quarter of the total memory capacity [8].

2.2 Accessing of Objects' Header

Java arrays are handled by the JVM as objects. Therefore the arrays are stored in the JVM heap in a layout that consists of the object header and the data, as shown in Figure 2. The total size of the object header is twenty four bytes, since TornadoVM is configured with the option of compressed ordinary object pointers (CompressedOops) as disabled. Thus, the first eight bytes in the header are used to

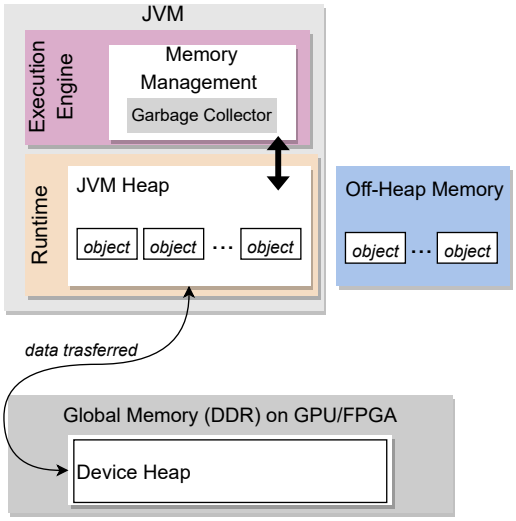

Figure 1: The memory management scheme between the JVM and TornadoVM (any version prior to v0.14).

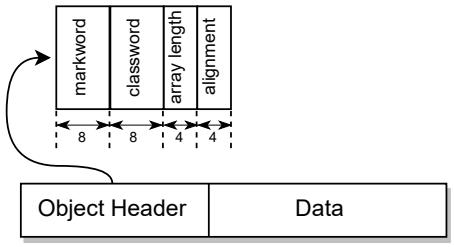

Figure 2: The layout of an object that stores an array in JVM when CompressedOops are disabled.

store the `markword` that stores meta information about the object, whereas the next eight bytes store the `classword`. The following four bytes contain the array length, while the latter four bytes are used to fix the alignment of data.

As mentioned in Section 2.1, the deployment of a device memory heap by TornadoVM is a mechanism that enables the generated heterogeneous code to access data in the same way as the JVM. Therefore, to access the data of an array and skip the object header, a code generated by TornadoVM has to add an offset of 24 bytes. For example, Listing 1 presents a code snippet that shows the format that TornadoVM uses to make the generated heterogeneous code access data. Although this snippet is presented in OpenCL C, the same mechanism is applied to the remaining TornadoVM backends (i.e., PTX, SPIR-V). In this example, the `object_base` is a pointer to the base address of the allocated object in the device heap memory, while `index` is the aligned value of the induction variable (i.e., `i`). As shown in line 5, the calculated `address` is used to load the data stored in the `i` position of the array. Similarly, line 7 shows how the generated kernel retrieves the size of the array from the object header.

Although accessing the header of an object from a kernel (e.g., OpenCL, PTX or SPIR-V) unlocks further functionality with regard to the implicit access of the array sizes, the alignment of the words,

```
1  // ...
2  index = i << 2;
3  offset = index + 24L;
4  address = object_base + offset;
5  data = *((__global int *) address);
6  // ...
7  array_length = object_base + 16L;
```

Listing 1: Example in TornadoVM to show the memory operations within a generated OpenCL code.

etc., it hinders the portability with other programming languages that do not adhere to the same memory layout. For instance, a kernel generated by TornadoVM to access the data by applying an offset of twenty four bytes for every load/store operation, cannot be executed in an ordinary C++ host application.

2.3 Replacement of Arguments with Literals

The TornadoVM JIT compiler is a superset of the Graal JIT compiler, extended with further compilation phases that specialize the compilation graph for execution on heterogeneous hardware. One of the optimization phases in the compiler is "constant folding". This phase is applied by the TornadoVM JIT compiler over the input arguments. The input arguments of a compiled Java method that hold a literal value are replaced by the compiler with a constant node. That optimization phase replaces all the occurrences of the input arguments with the corresponding constant value, thereby impacting the parameterization of the generated code. For instance, consider a method that uses an argument that stores the size of iterations to be internally processed by that method. If the size of the iterations is modified in a later stage of the Java program, then the generated heterogeneous code is not functional and the method has to be re-compiled.

2.4 Signature of Generated Kernels

The last challenge regards the signature of the generated heterogeneous kernels. The generated kernels contain TornadoVM-specific arguments which break their portability with other programming languages. A list of those arguments is as follows:

- `_kernel_context`: A pointer to a memory segment that is allocated to store information about the grid and the number of threads deployed by a kernel.
- `_constant_region`: A pointer to a memory segment that is allocated in constant memory, which is a region within the global memory that remains constant during the execution of a kernel [7].
- `_local_region`: A pointer to a memory segment that is allocated in the local memory, which is used similar to the CPU cache memory to store data shared between a specific number of threads (work-items) in a work-group [7].
- `_atomics`: A pointer to a memory segment that is allocated to be used for atomic memory operations.

3 CODE INTEROPERABILITY MODE

To enable the TornadoVM system to generate heterogeneous code (e.g., OpenCL kernels) that will be executable by any system out

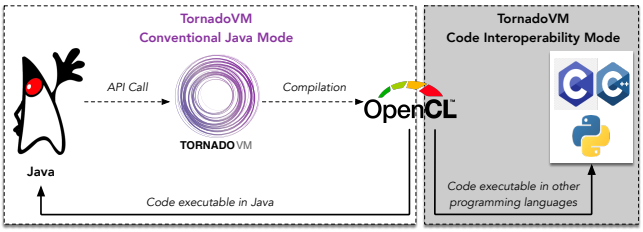

Figure 3: The Code Interoperability Mode in TornadoVM.

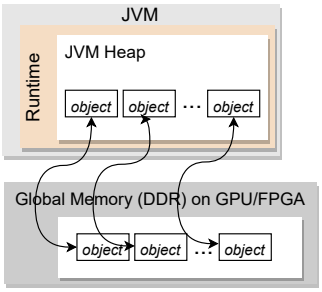

Figure 4: The memory management scheme of TornadoVM since version v0.14 and onwards.

```
1  private static void vectorAdd(int[] a, int[] b, int[] c,
2                                int size) {
3      for (@Parallel int i = 0; i < size; i++) {
4          c[i] = a[i] + b[i];
5      }
6  }
```

Listing 2: Example of a vector addition Java method implemented in TornadoVM.

of the JVM, we implemented a new mode, named "Code Interoperability Mode (CIM)". Figure 3 depicts how the proposed mode can unlock code interoperability with other programming languages beyond Java. The proposed mode contains numerous modifications in the TornadoVM JIT compiler and runtime as a mean to address the challenges described in Section 2. The following subsections present each modification.

To demonstrate each modification we present a Java implementation of a vector addition method in TornadoVM v0.14, as shown in Listing 2. The OpenCL C code that is generated by TornadoVM to execute in the conventional Java mode is presented in Listing 3. Listing 4 presents the OpenCL C code that is generated by TornadoVM using the CIM Mode.

Modification in the Memory Model of TornadoVM. To address the limitation of the maximum allocation size for a single allocation (Section 2.1), we have modified the TornadoVM memory management scheme to perform a separate allocation per input argument (i.e., array) of a compiled method, from version v0.14 and onwards. Figure 4 shows that the global memory of the device contains as many allocated segments as the number of objects (i.e., arrays) utilized by an accelerated method. The TornadoVM runtime is also augmented to manage the allocated memory segments in an efficient manner that would re-assign any pre-allocated segments that are not utilized. If the size of the required allocations

```
1  __kernel void vectorAdd(__global long *_kernel_context,
2                          __constant uchar *_constant_region,
3                          __local uchar *_local_region,
4                          __global int *_atomics,
5                          __global uchar *a,
6                          __global uchar *b,
7                          __global uchar *c,
8                          __private int size)
9  {
10     // ...
11     for(;i_4 < 1024;)
12     {
13         // BLOCK 2
14         l_5  =  (long) i_4;
15         l_6  =  l_5 << 2;
16         l_7  =  l_6 + 24L;
17         ul_8  =  ul_0 + l_7;
18         i_9  =  *((__global int *) ul_8);
19         ul_10  =  ul_1 + l_7;
20         i_11  =  *((__global int *) ul_10);
21         ul_12  =  ul_2 + l_7;
22         i_13  =  i_9 + i_11;
23         *((__global int *) ul_12)  =  i_13;
24         i_14  =  get_global_size(0);
25         i_15  =  i_14 + i_4;
26         i_4  =  i_15;
27         }  // B2
28     // BLOCK 3
29     return;
30 }  //  kernel
```

Listing 3: A vector addition OpenCL kernel generated by TornadoVM v0.14.

exceeds the available unutilized size, then the memory segments are released and new allocations are performed. The vector addition kernel that is generated by the Java snippet code in Listing 2 is shown in Listing 3. As shown in lines 5 to 8, the three input arguments that correspond to the Java arrays are pointers to the allocated memory segments that reside in global memory, while the last argument that corresponds to a variable that stores the size is allocated in private memory (i.e., stored in a register).

Replacement of Header Offset for Memory Accesses. As discussed in Section 2.2 and shown in line 16 of Listing 3, the TornadoVM JIT compiler emits an offset to skip the header of the array objects. Therefore, we modified the TornadoVM JIT compiler to eliminate this offset (See line 12 of Listing 4) in order to make the memory operations compatible with the memory layout of other programming languages that operate out ot the JVM.

Disabling the Replacement of Arguments with Literals. As shown in line 11 of Listing 3, the input argument of the size should be used as the bound value in the for loop, but it is replaced by the compiler with the literal 1024. To eliminate the replacement of input arguments that store literals when acting in the CIM mode, we modified the TornadoVM JIT compiler to disable constant folding for this case. Thus, the generated code utilizes the input argument for the loop bound value as shown in lines 7-8 of Listing 4.

```
1  __kernel void vectorAdd(__global uchar *a,
2                          __global uchar *b,
3                          __global uchar *c,
4                          __private int size)
5  {
6      // ...
7      i_3  =  (ulong) size;
8      for(;i_5 < i_3;) // Replacement of constant value
9      {
10         // BLOCK 2
11         l_6  =  (long) i_5;
12         l_7  =  l_6 << 2; ; // Header offset (24L) is skipped
13         ul_8  =  ul_0 + l_7;
14         i_9  =  *((__global int *) ul_8);
15         ul_10  =  ul_1 + l_7;
16         i_11  =  *((__global int *) ul_10);
17         ul_12  =  ul_2 + l_7;
18         i_13  =  i_9 + i_11;
19         *((__global int *) ul_12)  =  i_13;
20         i_14  =  get_global_size(0);
21         i_15  =  i_14 + i_5;
22         i_5  =  i_15;
23     }  // B2
24     // BLOCK 3
25     return;
26 }  //  kernel
```

Listing 4: A vector addition OpenCL kernel generated by TornadoVM with the Code Interoperability Mode.

Modification in the Signature of the Generated Code. The last modification is the elimination of the four TornadoVM-specific input arguments (see Section 2.4) from the generated kernels. As a result, the signature of the generated kernel in Listing 3 is similar to the layout of the compiled method in Listing 2. Note that although the type of the pointers differ, the kernel is compatible to operate with any type used in a host code (e.g., C++).

4 CONCLUSIONS

In this paper, we show our work-in-progress regarding enabling the cross-language interoperability of heterogeneous code that is generated by Java. To realize that concept, we have extended TornadoVM with a new mode, named "Code Interoperability Mode", which unlocks the portability of its generated codes to other programming languages and systems. In the future, we plan to assess the performance of the proposed mode across a large number of benchmarks in order to attest the impact of generating less number of lines of code as well as performing memory accesses without skipping the header of the objects.

ACKNOWLEDGMENTS

This work is partially funded by grants from Intel Corporation and the European Union Horizon 2020 ELEGANT 957286. Additionally, this work is supported by the Horizon Europe AERO, ENCRYPT and TANGO projects which are funded by UKRI grant numbers 10048318, 10039809 and 10039107.

REFERENCES

[1] AMD. 2022. . https://aparapi.github.io/

[2] Florin Blanaru, Athanasios Stratikopoulos, Juan Fumero, and Christos Kotselidis. 2022. Enabling Pipeline Parallelism in Heterogeneous Managed Runtime Environments via Batch Processing. In Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE).

[3] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. 2018. Exploiting High-Performance Heterogeneous Hardware for Java Programs Using Graal. In Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang).

[4] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic Application Reconfiguration on Heterogeneous Hardware. In Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE).

[5] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE).

[6] Juan José Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. 2015. Runtime Code Generation and Data Management for Heterogeneous Computing in Java. In Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ).

[7] Khronos OpneCL Working Group. Accessed in December 2022. SYCL™ Specification: Generic heterogeneous computing for modern C++. https://registry.khronos.org/SYCL/specs/sycl-2020-provisional.pdf

[8] Khronos OpneCL Working Group. Accessed in December 2022. The OpenCL C Specification. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html

[9] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. 2015. Compiling and Optimizing Java 8 Programs for GPU Execution. In 2015 International Conference on Parallel Architecture and Compilation (PACT). https://doi.org/10.1109/PACT.2015.46

[10] Lawrence Miller. Accessed in December 2022. GraalVM for Dummies eBook. https://go.oracle.com/LP=105746

[11] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All (Onward! 2013). https://doi.org/10.1145/2509578.2509581